A Sheet Music Organisation System

Interim Report

Submitted for the BSc in Computer Science with Industrial Experience

January 2015

by

Charlotte Godley

Contents

1 Introduction

Notation of western classical music has used a combination of the diastematic and phonetic notation (Rastall 1982) since the advent of Gregorian chant around 640AD (Taruskin 2005). The function of this notation falls into two main divisions: the expression of relationship in sound frequency, and the expression of relationship in time, or measure (Warner 1929).

The key element of this form of notation is the staff, a grouping of five horizontal lines. This is important as each line or space in the staff indicates a different sound pitch, a term meaning the relative "highness" or "lowness" of the sound (Land and Vaughan 1978).

This staff is divided by barlines, vertical lines delineating grouped units of sound and silence (formally referred to as notes and rests), which provides an indication of the unit's relationship in time by its juxtaposition to other groupings in the composition. These groupings are called measures or bars, with each bar having a variable maximum of notes and rests.

The representations of other parameters in staff notation are normally phonetic, such as indications p - meaning piano, or "quiet" - and pizz - meaning pizzicato, or "plucked" (Rastall 1982). This mechanism is complex in nature, and has a large but finite set of symbols which control every element of the composition.

A musician will often organise pieces notated using this system, or scores as they are formally known, in a music cabinet or filing cabinet (Sullivan 2012). These range from basic shelving to the more traditional large ornate cabinets where drawers only allow the user to see the top document, making it difficult to find scores which are deeper in the cabinet. This can often mean that music a performer will use regularly is kept on music stands and in piano stools, because if they were to return an item back into the music cabinet, it is likely that they would never find it again (Feist 2012).

This makes it difficult for instrumentalists to find a good way to organise their collections - this particular physical method allows for only one or two ordering choices, whereas the described notation system has many ways in which a piece can be identified.

It would be useful to a variety of musicians to be able to organise and search their collections using more than one mechanism. Such mechanisms have been developed in software automatically to cater for song title and composer name, but only allow for more detailed organisation by asking the user to provide more complex meta information (iMobilTec 2013). In a physical system this type of organisation would require file duplication. This project aims to create a sheet music organisation system for virtual music organisation, which will provide an automatic mechanism for the described problem.

This document discusses the aims and objectives of this project, background setting the project in a technical perspective, designs for the development of the solution, and a review of progress and project management to date.

2 Aims and Objectives

2.1 Project Aim

The overall aim of the project is to design and develop a sheet music library application, with the ability to organise and view personal sheet music collections, and download sheet music from the internet. Time permitting, it should also be able to generate sound from the sheet music, and import editable music from flat images.

2.2 Primary Objectives

The following objectives are of the highest importance to the project, and are a measure of whether the project has been completed.

- **Rendering of Musical Files**
  It will be necessary to render one or more formats of commonly stored musical files, as the aim of the project is to enable users to view and organise their sheet music collections.
- **Extraction of Metadata**
  The project will be required to extract important information from each piece, ranging from the simple nominal data such as title, composer, lyricist, to the more complex notation such as clefs, key signatures and meters used.
- **Connection to Online Music Collections**
  It should also be possible to connect to online music collections, as it would be beneficial to users to be able to search and add to their collections using the same interface.
- **Develop the project using Test Driven Development**
  The project should be developed using Test Driven Development, as due to the complexity and amount of symbols required for music production, it is important that each feature and symbol be tested meticulously in order to ensure validity.

2.3 Secondary Objectives

The following objectives are to be completed if the primary objectives are met, and as such do not dictate the success or failure of the project, but rather are features which would add value.

- **Audio playback**
  It would be useful to a cross section of users to be able to play music files as sound clips, enabling the automatic creation of accompaniment parts, amongst other benefits.
- **MusicOCR conversion of images to parseable MusicXML**
  It would be easier for musicians to merge their physical and virtual music collections for automatic organisation if the solution provided a way to import flat, scanned sheet music into knowledgeable music files.

3 Background

3.1 Problem Context

This problem's main focus is the difficulty of organising classical sheet music, and how this can be made easier by the automatic extraction of key pieces of information. In order to understand what a performer may want to know about a particular piece, it is important to have a brief understanding of the elements of musical notation common to all compositions.

3.1.1 Clefs

As mentioned in the introduction, an important part of musical notation is a sound's frequency relation, denoted by the staff lines and spaces.

In this system, sound frequencies, or pitches, are denoted by letters A-G - each set of these eight letters is an octave, after which the next pitch above it will be the start of a new octave.

In order to provide a link between the lines and spaces of a staff and pitch name, a clef symbol is necessary:

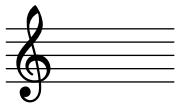


Figure 1: A staff with a treble clef

Each clef symbol denotes a different pitch name - in the above example, a G. The center around which this symbol is drawn - here, the second line from the bottom of the staff - indicates that this line or space will be known as a G. From this the reader can infer all other pitches by counting through the letters of the cyclic octave system, so in the given example, the pitch above becomes an A, and the pitch below becomes an F.

This symbol is important to a musician as different clefs are used to position the majority of the pitches in a piece on the staff, as this makes it easier to read. From this a performer can infer the average range of a piece, and predict whether this will be comfortable for the performer's chosen instrument or voice.

3.1.2 Keys

A second important indication to the player is the key, denoted by a key signature:

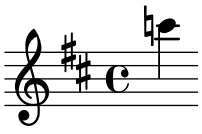


Figure 2: A staff with a key signature

A collection of symbols at the beginning of the piece indicate which pitches should be raised by half pitches, and which should be lowered. Raised pitches are called sharps, indicated by the # symbol, whilst lowered pitches are called flats, indicated by the ♭ symbol. Each key, which has a letter name and key type, has a different combination of flats or sharps.

This is a useful piece of notation to a musician as pieces in less common keys, such as C# major or F# major, may prove more difficult for the user to perform, and therefore they may want to filter out pieces in these particular keys. Similarly, in the case of singers, a singer's range may sit comfortably in one or two keys and they would perhaps want to find pieces in only these keys.

3.1.3 Meter

The third symbol denoted at the beginning of a measure is the meter, two numerals positioned like a mathematical fraction:

Figure 3: A staff with clef and 2/4 time signature, or meter

The most common meter is 4/4, sometimes denoted by a C indicating "Common time". The upper number of a meter symbol indicates the amount of beats in the bar. A beat simply refers to a note or rest, and the type of beat is indicated by the lower number. In this case, 2/4 indicates a measure will contain 2 crotchets, or quarter-length notes.

This information is important as it tells the performer how the rhythm and beat of the piece should be felt, counted and performed, and is useful for searching purposes as different meters, or time signatures as they are sometimes referred to, give the piece a different feeling, dictating the sort of occasion this piece would accompany.

For example, 2/4 is commonly used for march pieces, 3/4 is commonly used for waltzes and dance pieces, and 6/8 gives a similar, but more syncopated feel of a dance like piece.

3.1.4 Tempo

The speed of a particular piece, or the tempo, is indicated by an equation:

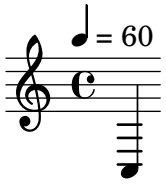


Figure 4: A staff with tempo marking

As explained above, this equation shows that the piece should be played at 60 beats per minute - the symbol dictating the sort of beat per minute depends on the time signature, here a crotchet (or quarter note) is given as the piece is in 4/4 time. Sometimes, this will be accompanied by a text direction to indicate speed or style, such as Andante, indicating a walking speed.

This indication would prove a useful identifier as pieces of different tempos provide variation in performance lists, so a concert organiser may want to find pieces with a variety of tempos.

3.1.5 Further metadata

Aside from these symbols, there are some items of textual information useful to the user.

The first of these would be the parts in the piece and their transpositions. Parts would be relevant as a particular group of instrumentalists may need parts that fit their instruments. If this is not the case for a given piece, however, a part written for a different instrument, for example, the Alto Saxophone rather than the Tenor Horn, may be compatible with the instrument anyway, if the transposition matches the instruments together.

Further to this, the user would want to know the piece's title, and names of publishers, composers, arrangers and lyricists of the work. Further to the composer name, it may be useful to know the date of composition as an indication of the piece's stylistic era, such as Classical/Baroque/Romantic, though this would not always be written on the sheet music so may need to be researched using the internet.

3.2 Comparison of Technologies

3.2.1 Programming Language

This project could be developed with a variety of programming languages, as displayed in the following table:

Language	Speed of development	Developer's Knowledge	Most recently used	Cross compatible
C#	Fast	A lot	2nd year	Yes
Python	Fast	A lot	In constant use for over a year	Yes
C++	Slow	Average	2nd year	Yes

The four key elements the developer is focussing on are speed of development, as the time constraint of a year means it is important that development is not hindered by the language itself, developer knowledge and most recent usage of the language as this will provide an additional time benefit, and cross compatibility, due to the different operating systems the developer intends to use in the course of development.

The developer understands C# is cross compatible through the use of the Mono Project, which is feature complete to C# 10(The Mono Project 2015), or Xamarin Studio and other such tools, but has not developed any applications with c# for multiple operating systems. For this reason, the developer feels more comfortable using Python, owing to the experience of writing applications for Linux and Windows in previous projects.

Due to these factors, Python has been selected, particularly as the developer used much of the features of python during an industrial placement and feels this is the most appropriate language for rapid development.

Further to these benefits, there are many projects in the field of musical software research currently in existence using this language, (The Python Foundation 2015) which will help when trying to debug issues and build upon previous research.

As a further technical challenge, the developer has elected to use Python 3, as this is the most up to date version, but also a version the developer has yet to work with and may requre some relearning of specific areas of syntax. This presents a technical challenge as many other open source projects and packages use version 2.7, so interfacing with other works may require submitting pull requests to update them to version 3.

3.2.2 File format

The project will require at least one default format for it to process music, which needs to have detailed information about what the score contains. The table below describes the options considered by the developer:

Format	Purpose
muscx	MuseScore notation
SIB	Sibelius notation
new format	this project only
MusicXML	sharing music between software

The first two options, muscx and SIB files, are formats used by the open source notation software MuseScore(MuseScore 2015a), and the world's most popular proprietary notation software, Sibelius(Avid 2015). Using either or both of these files would mean the majority of users would be able to use the application. However, both options couple this project with those particular packages, when users could still choose other software to write music with. Furthermore, the formats are specifically designed for those software packages and may have nuances which make development for this project more difficult. Additionally, Sibelius is proprietary so borrowing their file format may cause copyright issues.

The third option is to create an entirely new format. This would mean the file format was designed to the specification of the developer and therefore be entirely customisable and extensible. However, this project is created with the intention of organising, not composing music, so the files would have to be created or imported from other software packages manually if the project does not include composition. Therefore, this is the least applicable option.

The fourth and final option, which the developer has decided to implement, is MusicXML, a file format intended for sharing and archiving the world's sheet music(Make Music, Inc 2015). This particular format is used by a wide variety of software packages(Make Music, Inc 2015) and is included in the formats usable by both MuseScore(MuseScore 2015a) and Sibelius(Avid 2015), therefore neither couples the format with a program nor requires manual creation and import of current music files. However, this particular format was designed by a third party, and might therefore present a further technical challenge in learning how the format notates everything, which won't necessarily be the best method in the opinion of the developer.

3.3 Comparison of Algorithms for Rendering and Organising Sheet Music

3.3.1 XML parsing algorithms

For the rendering of sheet music, it will be necessary to parse a musicXML file into a hierarchy of objects, beginning with the overall piece and descending into each part and measure.

For the parsing of XML itself, there are two potential built in methods to choose from. The first loads the entire XML file into memory and provides methods for the developer to search the loaded file for specified tags. The developer has used this before in personal and industrial projects, and believes it is cumbersome to manipulate data in this way. Furthermore, the developer is focussing on rendering the information rather than rendering it with precise formatting, and many software packages implant musicXML files with very complex formatting information which may or may not be necessary.

The second option is using a different api called the Simple API for XML (SAX). In this method, the program loads the XML file tag by tag, and connects to call backs when specified things occur in the file, for example a new tag or piece of data inside tags, or the closing of an old tag. This is easier to work with as the developer can iteratively build up a collection of methods to handle each tag, and is better for memory management as only tags which are necessary to the project will have any effect on the object structure. For these reasons, this method has been selected.

3.3.2 XML verification algorithms

For both the algorithm options discussed in the above section, a further choice is whether to verify the XML parsed, using an online file validator, or presume the file is written in valid MusicXML.

The usual choice is to verify all XML, and is therefore the default option for both methods of parsing. Whilst this confirms that XML is valid before starting parsing of a file which could be corrupt, the speed at which files will parse is greatly reduced according to the speed of the user's internet connection. Furthermore, if the user is browsing their own music collection, it should not be necessary for the user to be connected to the internet.

Due to speed and functionality considerations, the choice has been made that the XML parser algorithm will not verify XML being converted to objects, or being examined for metadata. Given that most musicXML will be produced automatically by other programs, it is unlikely files opened by the project will be corrupt, though necessary steps will be taken to avoid this causing a problem in the program.

3.3.3 Metadata algorithm

The metadata algorithm has been designed so that, for a given folder, the program will parse all of the files with the XML extension for a given selection of information (for example, composer, piece title, instruments) and store this information in memory.

This is to be indexed either by the information title - e.g "composer"; or by the information itself - e.g "bartok". This will facilitate faster searching of the database for use when the user is finding a particular piece, and facilitate the production of auto generated playlists by the system. Depending on the method of indexing, the alternate indexer should be stored as part of the value in a key value pair format, alongside the file in which it was found.

It has been decided that in memory, this will be structured using a generic type. This decision has been taken because only 3 pieces of information per item of data will be stored, one of which will be the index of the data, so using a dictionary holding tuples would be more appropriate than creating an object as it simplifies the algorithm and therefore the debugging process.

Further to this, of the two xml algorithms described in section 3.3.1, the algorithm will use SAX, as this algorithm does not require all of the data and would therefore waste a lot of memory by loading in the entire file to memory.

3.3.4 Rendering Algorithm

The program is required to take the object structure and transform it, in some way, to musician readable sheet music.

This could be achieved using an entirely new algorithm of the developer's design, with the output going directly to the render window using different glyphs and fonts extracted from their relevant classes.

However, using fonts may cause difficulty in creating functionality such as panning and zooming, features which are common to most render window applications. Furthermore, the conversion of even basic sheet music to a readable format would require a high level of precision and complexity, and creating a new algorithm would be considered reinventing the wheel, so to speak, as this is a process that has been covered by many different applications (like MuseScore(MuseScore 2015a), Finale(Make Music, Inc 2015) and Sibelius(Avid 2015)). Lastly, the process of debugging whether the symbols are correct would require visual checking by the developer and would be difficult to debug automatically.

It would be possible to alleviate the panning and zooming problem by converting the typesetted symbols to an image or PDF file and using a built in image rendering library, such as wxPython(The wxPython Team

2012). However, this method still involves reinventing the wheel and problems with visual debugging being required.

Considering these factors, it has been decided that the developer will be using a third party typesetting system known as Lilypond. Lilypond is a typesetting language and system developed to typeset the highest quality sheet music(Lilypond 2015), which takes an input file and outputs a PDF or image. As this is a language unto itself and has been in development for many years by the Open Source community, this will alleviate the problem of visual debugging - the developer can instead create a formatted lilypond output for each class based on its attributes and confirm that this is as expected, which can be done using unit tests.

3.4 Comparison of Technologies for Importing Online Musical Sources

3.4.1 Musical Sources

Two open and free sources of sheet music have been selected for potential inclusion, which will enable users to connect their own music collections with new music without using a browser to peruse collections. The first is **MuseScore Online**, which is a community website created for composers to upload share and discover compositions using the MuseScore platform (MuseScore 2015b).

This has been selected due to the number of files available, the openness of the platform and the well documented API created by the developers. It will, however, be necessary to manage copyright issues, as pieces published on this website may be published under the license of the composer's choosing and therefore may cause issues with certain types of users, in particular those performing commercially.

The second selected source is the **IMSLP**. This is the **International Music Score Library Project**, built with the intention of sharing the world's public domain music and contains 290,000 scores to date (IMSLP 2015). This may be a questionable source, as not all pieces are uploaded in MusicXML format due to the pieces being scanned and uploaded by community members, rather than being automatically generated by a piece of software. However, this source does not raise any copyright issues as all pieces are no longer covered by copyright.

It may also be possible to import collections from subscription services and websites enabling purchase of music, such as **MusicNotes.com**. However, this will require closer contact with the companies maintaining the website and may not be appropriate for an educational and academic purpose.

3.4.2 Searching Algorithm

The APIs for both selected sources provide a variety of output formats, the 2 most prominent being XML and JSON. The algorithm for browsing the source from the program will need to in some way, contact the server to confirm whether there are pieces which have a specific attribute entered by the user, and download the file if the piece is selected.

It would be possible to use an algorithm which repeatedly connects to the API and polls for the relevant input from the user, returning a list of options which the user would then select from and download from the server. Whilst this would be simple to implement, this would make the program considerably slower, as it would require repeated connection to the internet. This would also cause problems for the maintainers of the server, as repeated requests from a piece of software would cause a heavy load on the server.

It has therefore been decided that the software will cache a copy of all metadata served from each online source, and search for the relevant inputted data from this, and then, if necessary, collect the relevant file from the server. This would require a connection to the server only twice - once when updating metadata sources, and once when downloading a file - rather than a persistent or repeated connection.

3.5 Comparison of Algorithms for Sound Output and Image Input

3.5.1 MIDI algorithm

The sound output algorithm must, for a given part or selection of parts, output the sheet music to a MIDI or MP3 file, which can then be played within the program.

It has been decided that each class in the solution will have a method to produce this output. Each class holding objects as members will call each sub object's sound output method, which will be saved by the program as a full outputted file and played using a python sound output library.

This provides an extendable architecture, as the developer could choose to produce different outputs, such as one for MIDI and one for MP3, which would be called in the same way.

3.5.2 Image input algorithm

In order to import images or flat files into the chosen file format, it will be necessary for the program to include the ability to apply music optical character recognition to the file, and save the output to MusicXML, which can then be parsed by other parts of the program.

Creating a new algorithm It would be possible for the developer to produce their own algorithm for converting new imported images into the chosen file format. This would allow the developer to optimise the algorithm according to their own wishes, and provide sufficient technical challenge.

However, this project is concerned with music organisation, not OCR specifically, and as such the project is too large to commit a sufficient amount of time to this particular algorithm in order to make it function as well as other algorithms.

As a reference point, Optical Character Recognition for natural languages has taken may years to develop and perfect, and has been an attractive research area and idea to a wide variety of users(Fujisaki et al. 1990). OMR, or Optical Music Recognition, has been the focus of international research for over three decades, and while numerous achievements have been made, there are still many challenges to be faced before it reaches its full potential(Bainbridge and Bell 2015).

Using third party software It has therefore been decided that OCR as a topic is too large for this project, and if this goal is included in the project, it will be through communication with other systems, such as Audiveris, an open music scanner (Audiveris 2015).

This removes the technical challenge of producing an entirely new algorithm, but adds the challenge of understanding how optical music character recognition scanners work, and how they can be integrated with the system, particularly if the third party package is not developed in Python.

3.6 Alternative Solutions

The majority of music software available to the masses covers the creative process of writing music, with the two most well known packages being the aforementioned Sibelius, a commercial software package costing around £500 (Avid 2015), and the freely available and Open Source MuseScore(MuseScore 2015a).

4 Designs

4.1 System Design

4.1.1 Class diagrams and mind map

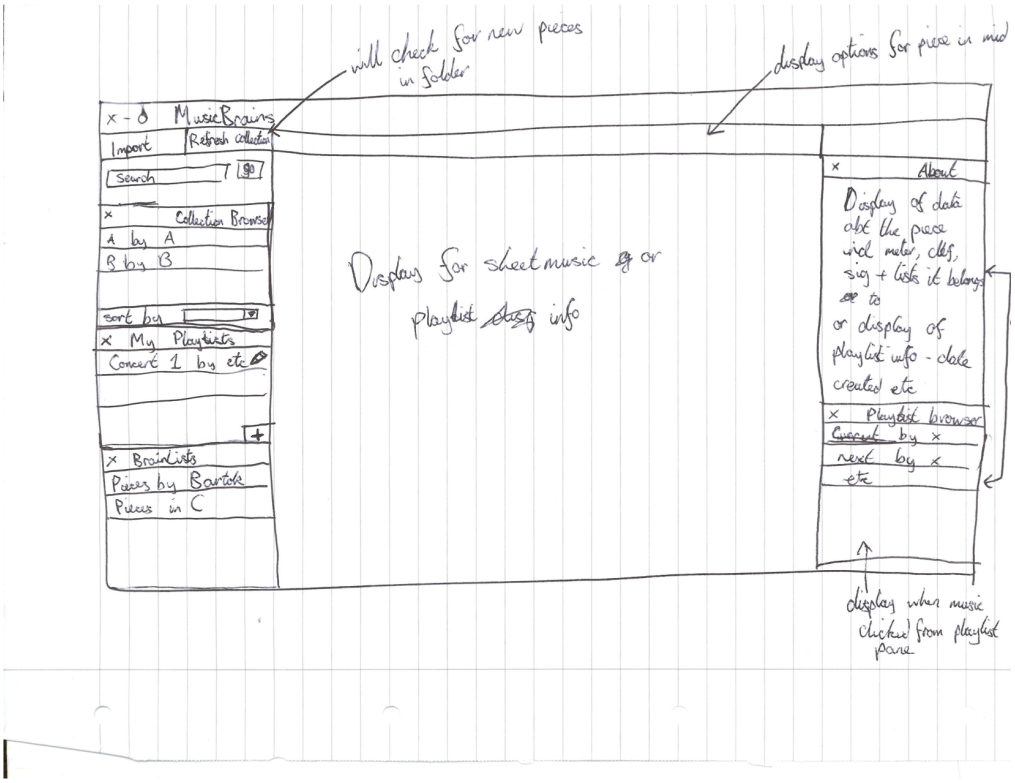
The developer used a mind map to initially appraise the connection between each musical symbol. This helped break down a piece from a musician's view, and showed what information would be necessary between each symbol's class.

An initial class diagram was drawn from the mind map. This was modified during the course of development testing and research of the initial model. In particular, the developer looked at other sources such as Music21, a toolkit for computer-aided musicology(MIT 2013), which helped the developer to examine whether the initial model was missing any classes or attributes. The diagrams described are in the appendices.

4.2 UI Design

The User interface for this project is designed with the average musician in mind, and the interfaces that user would normally have used. With this in mind, the designs explained below and shown in Appendix D are inspired by other music applications in common usage.

4.2.1 Main Display



The above image shows the main view of the application. Various panes to the left can be closed using the X button and show different ways the music can be displayed, either as individual units or as playlists. The panes to the right display content based on what is selected for display in the middle, larger window, generally

either a playlist or a singular piece of sheet music. Updates to these panes and various pop up boxes associated with different buttons in the display are in the appendices.

4.2.2 Musician feedback survey

In order to understand how well this user interface works with a variety of users, a survey was designed which will be given to a selection of musicians, who will feedback on how easy the UI is to use and any updates which should be made to improve it. This feedback session will be performed after the initial sketches are made into a virtual user interface with no back end connected to the buttons. An example survey is provided in the appendices.

4.3 Test Design

The developer has chosen to use test driven development. This is an agile software development methodology which utilises the rules that the developer should never write a signle line of code unless there is a failing automated test(Newkirk and Vorontsov 2004). This methodology has been chosen as the nature of the notation of music means that meticulous detail must be payed to how and with what symbol every element is notated, and Test Driven Development will ensure that every detail has been validated and proven to work.

The list of tests to date, which are written on an adhoc basis as the developer builds up functionality, are attached in the appendices, along with a description of current testcases used to confirm the functionality works with real world examples of sheet music.

5 Project Management Review

5.1 Current progress

During the initial stages of the project, the developer has spent time researching the appropriate language to use, the file format and the methods and algorithms used by other packages. This involved looking at the projects done in music in Python previously, such as Music21 (MIT 2013), and other projects listed on the Python Foundation website(The Python Foundation 2015).

Many important decisions were made from this research period, such as the decision to use Lilypond to typeset music files rather than create a new algorithm and the research of MusicOCR options available, leading to the decision that MusicOCR is too big a topic for this project to create a new algorithm.

After this research period, the developer began creating class diagrams and putting in some initial code implementation for the rendering and metadata objectives. It was decided after this intial stage to use Test Driven Development, as the code base and algorithm for loading in a music file was becoming hard to confirm crucial details were being parsed correctly. A set of unit tests were written for the initial implementation, and from this point onward the developer applied the methodology.

To date, the developer has created a working MusicXML parser which loads a MusicXML file into a tree of objects, each one having a ToString override method for debugging, which enables the developer to confirm that each object has loaded the correct members. The developer has began working on the rendering portion, which involves learning the Lilypond syntax appropriate to each class. This forms a significant portion of the first objective of rendering sheet music. The developer also created an implementation of the metadata parsing algorithm, though since deciding to use SQLite for data caching, this portion still needs refactoring.

The developer has also created a set of user interface designs, and a spreadsheet of tests and their purpose developed to date, which are in the appendices.

5.2 Adjustments made

based on coursework deadlines issues with stress/multiple projects being handled review and modifications made, and future things to consider in project management based on these

5.3 Revised timeplan

decision on objective implementation: OCR

Appendices

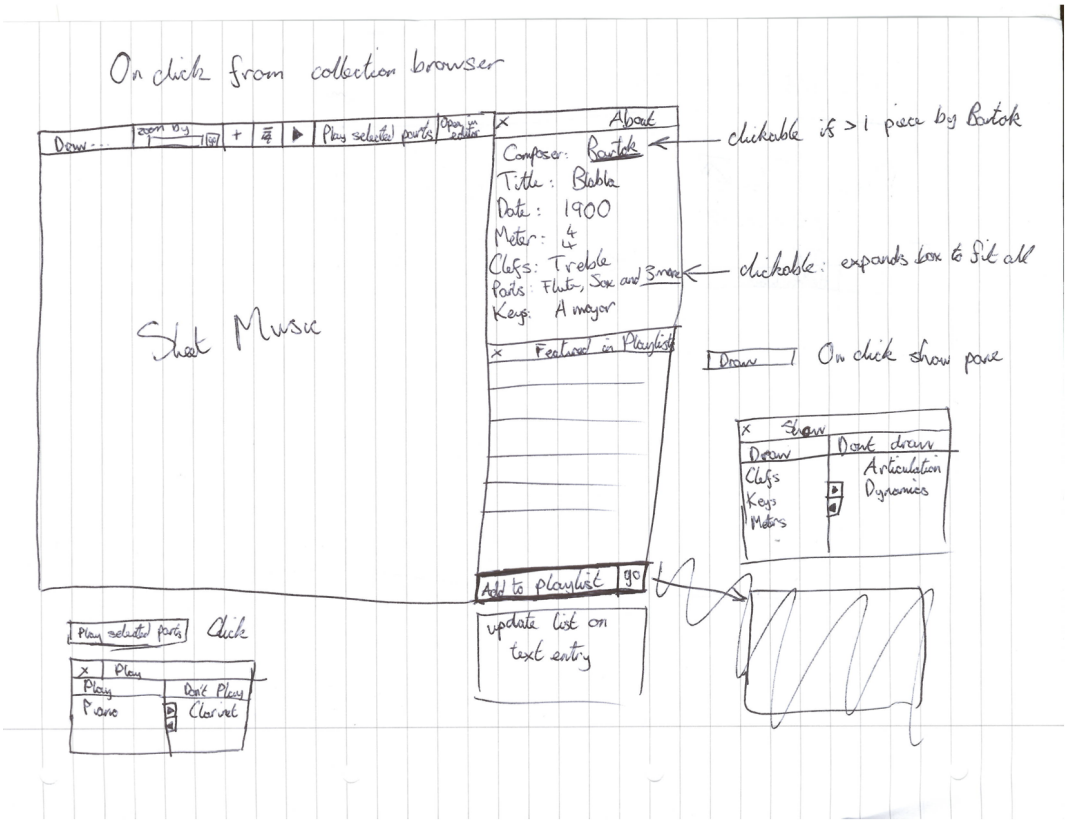
A Mind map of elements of Music

B Initial class diagram

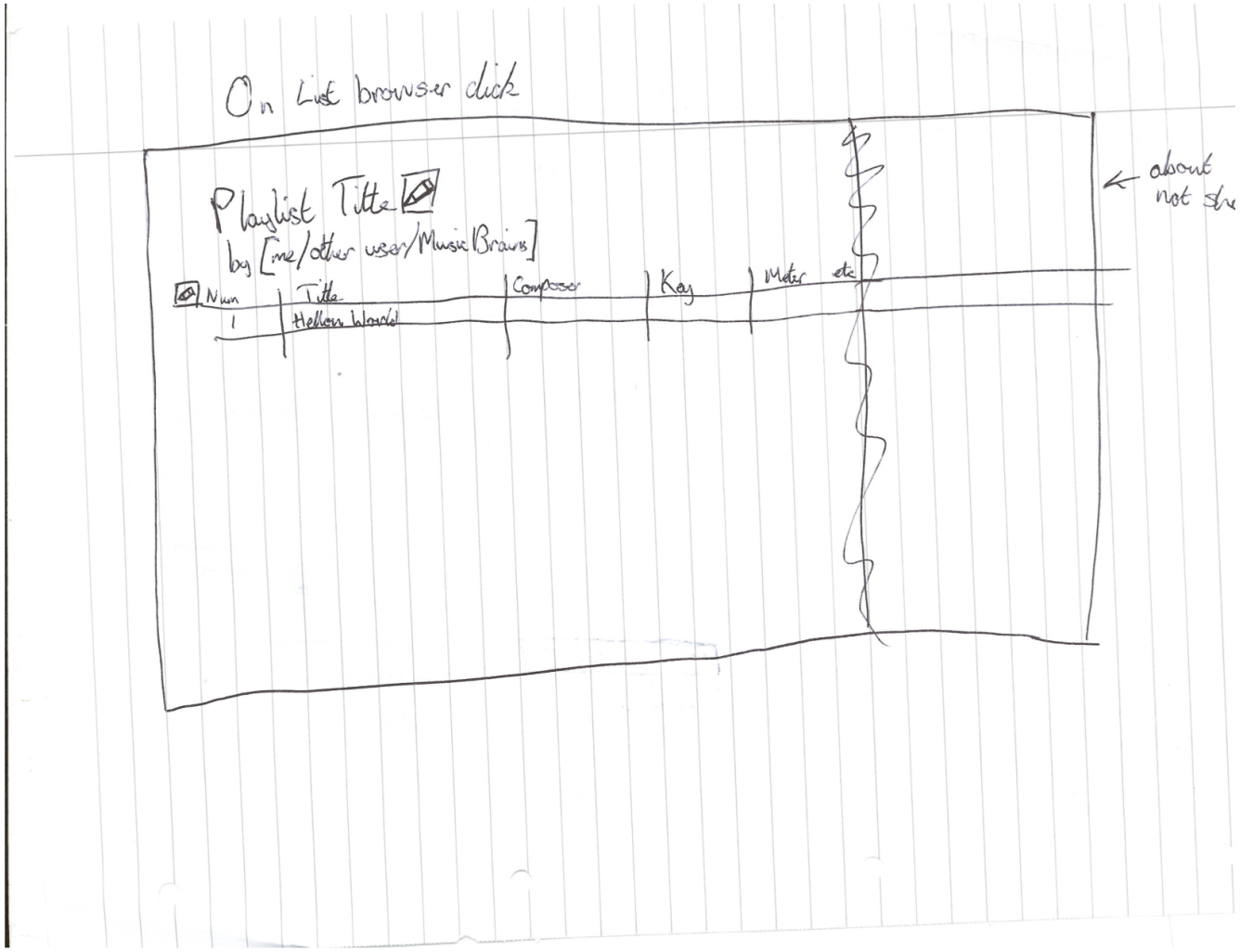
C Revised computerised class diagram

D Initial User Interface Design

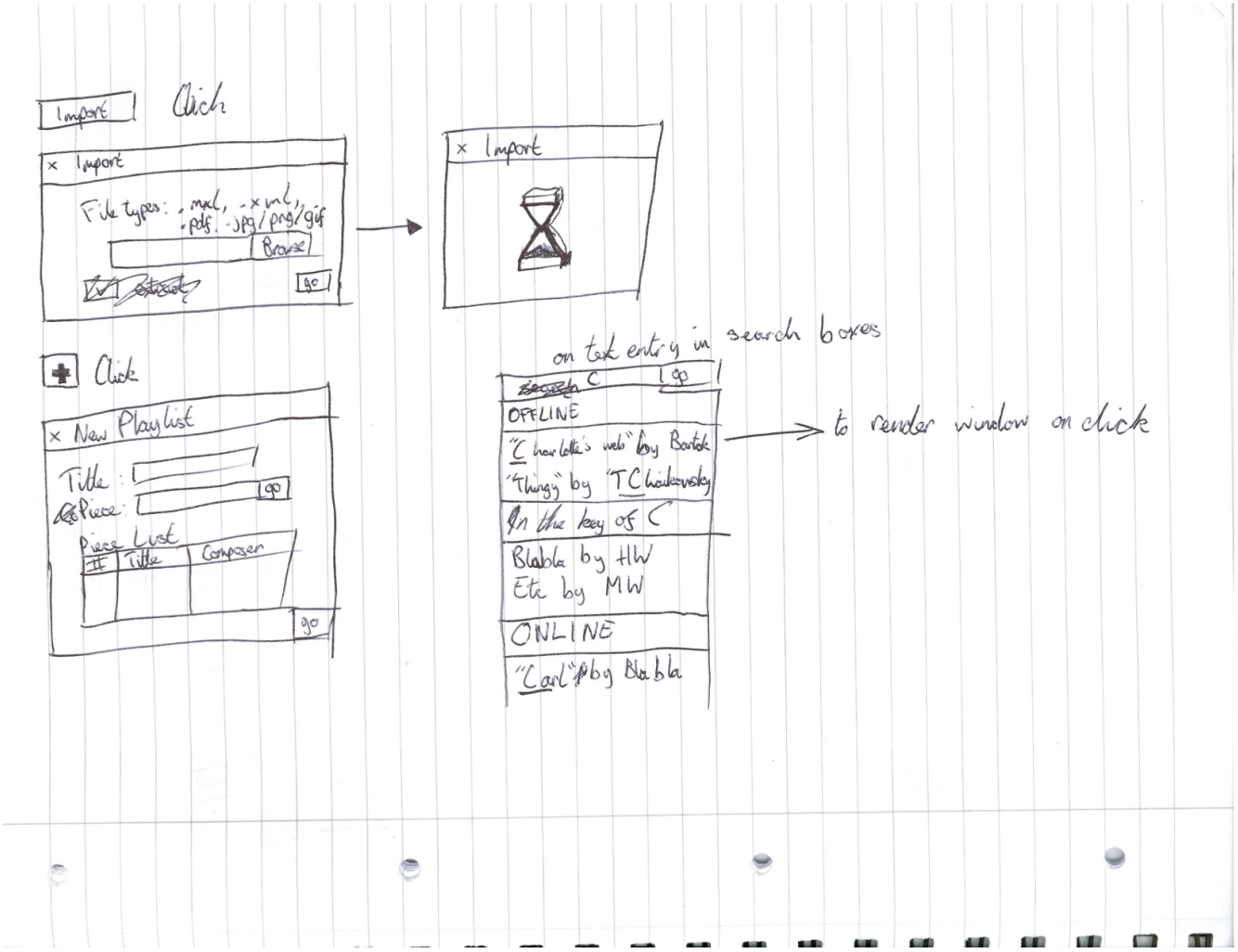
D.1 Changes to main display when sheet music selected



D.2 Changes to main display when playlist is selected



D.3 Other Panes and Popup windows



E User Interface Feedback survey

F Revised User Interface Design

6 References

Audiveris (2015). *Open Music Scanner*. URL: https://audiveris.kenai.com (visited on 09/01/2015).

Avid (2015). *Sibelius: the leading music composition and notation software*. URL: http://www.sibelius.com/home/index_flash.html (visited on 06/01/2015).

Bainbridge, David and Tim Bell (2015). *The Challenge of Optical Music Recognition*. URL: http://www.eecs.harvard.edu/~cat/cs/omr/docs/bainbridge-bell-challenge-of-omr.pdf (visited on 2001).

Feist, Jonathan (2012). *Sheet Music Cabinets: a rant*. URL: http://jonathanfeist.berkleemusicblogs.com/2012/08/07/sheet-music-cabinets-a-rant/ (visited on 06/01/2015).

iMobilTec (2013). *Calypso Jam - sheet music and fakebook organizer*. URL: https://itunes.apple.com/gb/app/calypso-jam-sheet-music-fakebook/id586636769?mt=8 (visited on 11/01/2015).

IMSLP (2015). *IMSLP/Petrucci Music Library*. URL: http://imslp.org (visited on 06/01/2015).

Lilypond (2015). *Lilypond - Music notation for everyone*. URL: http://lilypond.org/index.html (visited on 06/01/2015).

Make Music, Inc (2015). *MusicXML for Exchanging Digital Sheet Music*. URL: http://www.musicxml.com (visited on 06/01/2015).

MIT (2013). *Music21: A toolkit for computer-aided Musicology*. URL: http://web.mit.edu/music21/ (visited on 01/13/2015).

MuseScore (2015a). *MuseScore Tour*. URL: http://musescore.org/en/musescore-tour (visited on 06/01/2015).

– (2015b). *Sheet Music Sharing*. URL: http://musescore.com (visited on 06/01/2015).

The Python Foundation (2015). *Python in Music*. URL: https://wiki.python.org/moin/PythonInMusic (visited on 06/01/2015).

Fujisaki, T. et al. (1990). "Online Recognizer for Runon Handprinted Characters". In: *10th International Conference on Pattern Recognition* 1.

Land, Lois Rhea and Mary Ann Vaughan (1978). *Music in today's classroom: creating, listening, performing*. Harcourt Brace Jovanovich, Inc.

Newkirk, James W. and Alexei A. Vorontsov (2004). *Test-Driven Development in Microsoft .NET*. Microsoft Press.

Rastall, Richard (1982). *The Notation of Western Music: An Introduction.* St. Martin's Press.

Sullivan, Anne (2012). *Organize Your Sheet Music – What's Your System?* URL: http://harpmastery.com/organize-your-sheet-music-whats-your-system/ (visited on 01/11/2015).

Taruskin, Richard (2005). *The Oxford History of Western Music*. Vol. 1. Oxford University Press.

The Mono Project (2015). *The Mono Development Project*. URL: http://www.mono-project.com/docs/about-mono/languages/csharp/ (visited on 01/11/2015).

The wxPython Team (2012). *PDF viewer - wxPython (phoenix) documentation*. URL: http://wxpython.org/Phoenix/docs/html/lib.pdfviewer.html (visited on 01/11/2015).

Warner, Sylvia Townsend (1929). *The Oxford History of Music, Introductory Volume*. Oxford University Press. Chap. 3.